A Generator for Language-Specific Debugging Systems

Rolf Bahlke
Bernhard Moritz
Gregor Snelting

Fachgebiet Programmiersprachen und Übersetzer II
Fachbereich Informatik
Technische Hochschule Darmstadt
Magdalenenstr. 11c
D-61 Darmstadt, West Germany

Abstract

We present a system which generates interactive high-level debugging systems from formal language definitions. The language definer has to specify a denotational semantics augmented with a formal description of the language-specific debugging facilities. The generated debugger offers the traditional features such as tracing programs, setting breakpoints, displaying variables etc; interaction with the user is always on language level rather than on machine level. The concept has been implemented as part of the PSG-Programming System Generator, and has successfully been used to generate debuggers for Pascal and Modula-2. The core of the implementation consists of an interpreter for a functional language, which has been extended with the language-independent mechanisms needed in order to allow interaction with the user during program execution.

1. Introduction

It is generally accepted that program testing and debugging is an important phase during software development, and that therefore a programming environment should supply a debugging tool which helps users to detect and correct programming errors. Typical features of debugging systems include

- the possibility to display program status information like the values of program variables, the current position in a program, the current procedure/function nesting or a protocol of control flow
- the possibility to influence control flow, e.g. setting and resetting of breakpoints, tracing parts of a program , single-stepping program statements or triggering certain actions by certain conditions on the program status.

Several general requirements for debugging systems have also been formulated, including the following ([9]):

- The debugger must be interactive and should make use of full-screen or even bitmapped user interfaces
- It must be possible to debug on source language level, rather than on machine level (no hex dumps)
- For use with several different source languages, the debugger should on the other hand offer a common user interface across languages.

It is the aim of this paper to show how such a debugging system can *be generated automatically* from an extended specification of a language's semantics. The work described below has been done within the PSG project ([2]). PSG stands for *Programming System Generator* and produces interactive language-specific programming environments from formal language definitions. PSG environments are designed for programming-in-the-small. One of the subsystems generated by PSG is a context-sensitive hybrid editor, which is generated from the language's syntax and context conditions. A description of PSG editors can be found in ([3]). Another component is an interpreter which is generated from the denotational semantics of a language. It is characteristic for PSG language definitions that they are written in an entirely nonprocedural, formal specification language; there are no user-coded attribute evaluation functions or interfaces to standard programming languages.

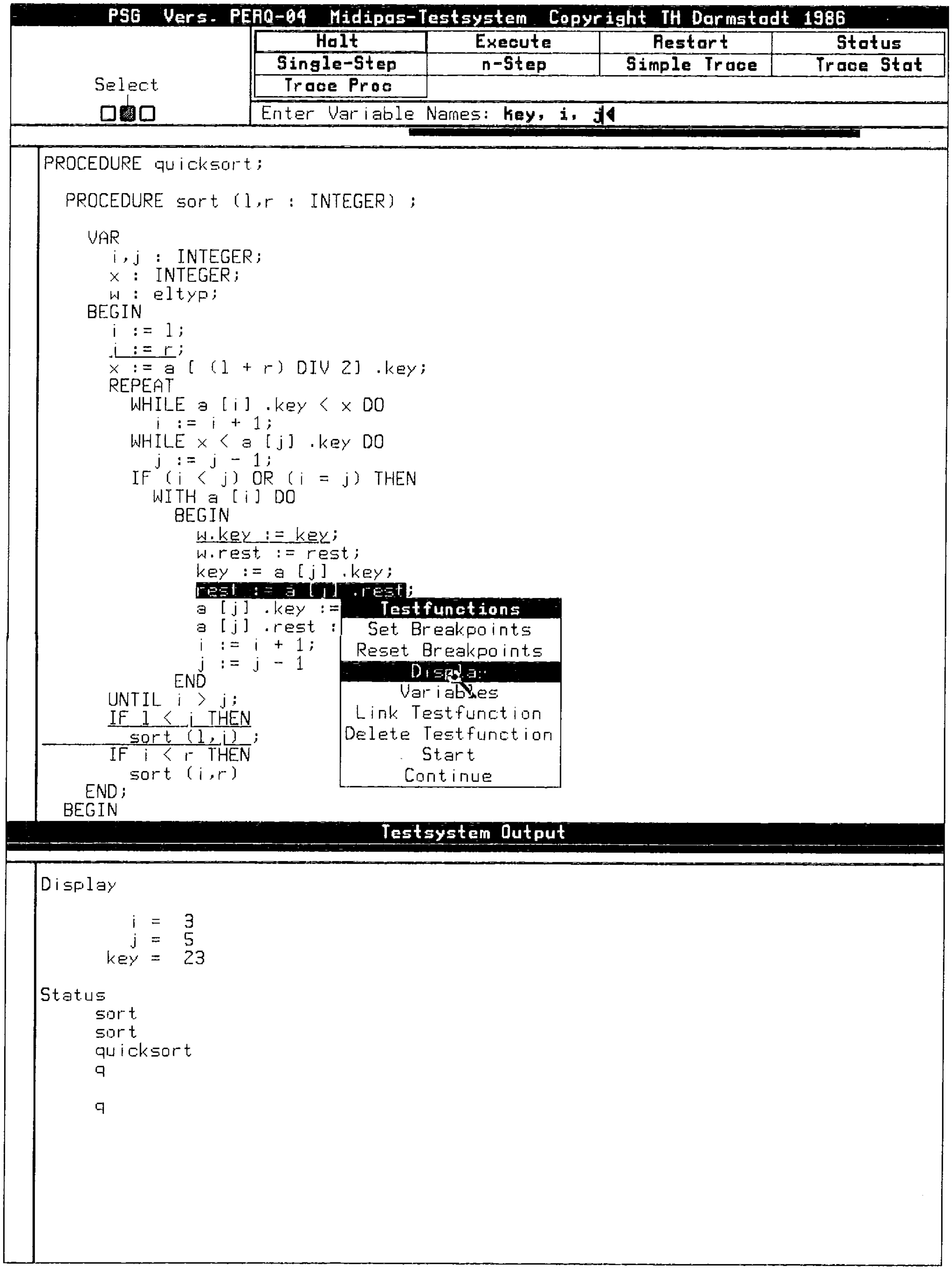
When PSG evolved, we felt the need to extend the interpreter (which has been part of PSG environments ever since) with interactive, high-level debugging facilities. The aim of undertaking this effort has been threefold: First, as stated above, it is important to have a debugger available within an interactive programming environment. Second, we wanted to see whether it is really possible to generate all language-specific parts of an environment, starting with the scanner and ending up with the debugger, from an entirely formal description of a language. Third, we wanted to have a tool which allows rapid generation of debuggers and can therefore be used for investigations on the ergonomic aspects of debugging within a language design lab.

The rest of this paper is organized as follows: First of all, a description of PSG debuggers is given. Next, we explain what the specification language looks like and what the language definer has to do. Third, we discuss several implementation details, especially the interaction between the interpreter kernel, the debugger and the rest of the environment. Finally we present some experiences we have made so far, this includes a discussion of the limitations of the current implementation.

2. Overview of the generated debuggers

The figure on the next page shows a typical screen within the Pascal debugger on a PERQ workstation. The screen is divided into three areas: the upmost window contains the so-called system menu (including a description of the current mouse button assignments, as well as a prompt and message area). The second window is the editor window and contains program text; the third window is used for program and test system output. The two large windows both have a scroll bar as well as a thumb bar attached which can be used for positioning. Within the program area, underlined text lines represent breakpoints; the inverted line represents the program piece which is executed at the moment. The system menu as well as the pop-up menu associated with the inverted program text offer the available debugging functions, as specified by the language definer.

Within the Pascal debugger, several trace options are available (execute [i.e. no trace], single step, n-step, simple trace, trace statements, trace procedures), which differ in the granularity of the tracing process: 'Single-step' will, during execution, invert the textual representation of all those nodes of the program's abstract syntax tree which are touched, whereas 'trace procedures' will invert only the procedure call statements. The trace functions also differ in their interaction behaviour: After a 'Single Step' or 'n-step', execution will be interrupted; 'Trace statements' and 'Trace procedures' will not interrupt



```
PROCEDURE quicksort;

  PROCEDURE sort (l,r : INTEGER) ;

    VAR
      i,j : INTEGER;
      x : INTEGER;
      w : eltyp;
    BEGIN
      i := l;
      j := r;
      x := a [ (l + r) DIV 2] .key;
      REPEAT
        WHILE a [i] .key < x DO
          i := i + 1;
        WHILE x < a [j] .key DO
          j := j - 1;
        IF (i < j) OR (i = j) THEN
          WITH a [i] DO
            BEGIN
              w.key := key;
              w.rest := rest;
              key := a [j] .key;
              rest := a [j] .rest;
              a [j] .key :=
              a [j] .rest :
              i := i + 1;
              j := j - 1
            END
      UNTIL i > j;
      IF l < j THEN
        sort (l,j) ;
      IF i < r THEN
        sort (i,r)
    END;
  BEGIN
```

```
Display

        i =  3
        j =  5
      key =  23

Status
     sort
     sort
     quicksort
     q

     q
```

execution. Trace speed can be adjusted by the user. It is possible to display additional information during tracing within the second window.

Execution is also interrupted if flow of control reaches a breakpoint or if the user presses a special function key. When execution is interrupted, the user may select among the available debugging functions which have been generated. He may change breakpoints, select functions which display information about the program status, he may link test functions to special places of the source program which will be executed when control reaches (or leaves) the selected part of the program, or he may initiate more complex test functions. The display functions available within the Pascal debugger can be used to display the values of the variables which are visible at the current breakpoint (the user will be prompted for the names of the variables to be displayed). Other debugging functions include counters for statements and procedure calls as well as a traceback of the dynamic procedure chain. It is even possible (though not part of the Pascal debugger specification) to trigger arbitrary complicated functions by user-defined conditions on the program status (Example: "if the value of a specific variable changes, then force an interrupt").

Since program and test system output are displayed in a second window, the user has always a complete overview of the execution history: he simply may scroll backwards within the second window. This feature (as well as scrolling within the program text) is always available if execution is interrupted. It depends on the definition of the debugging functions whether they will interrupt execution or not.

It is possible to execute incomplete programs: If flow of control touches incomplete parts (*placeholders*) within a program, the user will be asked to enter these parts using the editor. For reasons explained later, it is however *not* possible to change program text, flow of control or values of variables during execution.

3. What the language definer has to do

A debugging system is based on two concepts already used in PSG systems: the representation of programs as abstract syntax trees and the denotational definition of semantics. Therefore, the first step towards a debugging system is the specification of the denotational semantics of the language in question. [2] describes in some depth how this can be done within PSG.

The mechanism to set or reset breakpoints is language independent, thus there is no need to describe it in debugger definitions. All nodes of the program's abstract tree may have a breakpoint attached.

In order to describe how to implement counters for e.g. statements or procedure calls, *nesting functions* may be used. It is enough to enumerate the nodes of the abstract syntax which influence these counters in order to assign language-independent algorithms to these nodes and to organize a stack of e.g. procedure calls during program execution. Such a stack can also be used for a traceback.

During execution, the relevant program status information (actual symbol table, values of variables etc.) is hidden within variables within the semantic equations. It is therefore necessary to describe how to access the actual values of these variables. This can be done by specifying the names of these semantic variables together with the names of the abstract tree nodes where their values change or a new instance of such a variable is created. In the definition of the Pascal debugger these *select functions* look like:

```
SELECT
   env  : STACK    block;
          UPDATE   procdecllist, vardecllist,
                   typedecllist, constdecllist;
   state : UPDATE  statement;
END SELECT
```

In Pascal, the meaning of a statement is a function which maps environments onto state transformations, whereas the meaning of a declaration is a function which changes environments (we deliberately omitted the GOTO statement and therefore could use a direct semantics without continuations). Therefore, the above specification says that a declaration will change the environment (for variables, this is a mapping from variable names to locations), whereas the execution of statements will change the state (which is a mapping from locations to values). Select functions are a mechanism to make variables from the semantics definition visible in the debugging specification, these selected variables can be used in the rest of the debugger specification.

In order to describe what the user actually can do during debugging, the language definer has several classes of build-in debugging concepts available, which are called basic functions and differ in generality and complexity. Two simple cases of basic functions are *display functions* and *trace functions*. The task of a display function is to analyze the selected program status information (as specified above) and write the result onto the screen. It will be invoked by selecting its corresponding menu item. Trace functions will be evaluated when the debugger is in trace mode. They are linked to nodes of the abstract syntax tree. For each trace function, the language definer has to specify their granularity, whether they will interrupt execution or not, and which information should be displayed during tracing.

Each basic function must be specified as an auxiliary function written in the semantics definition language, which is an extended type-free lambda calculus. The application part of the debugger definition contains the information about the parameters which should be passed to the auxiliary functions. Example:

```
APPLICATIONS
 DISPLAY FUNCTIONS
   display:       (vardisplay env, state, INPUT);
 END DISPLAY FUNCTIONS
 TRACE FUNCTIONS
   tracestat:     statement ->(trace SOURCE);
   traceassign:   assign    ->(changedvar env,
                         state, RESULT, SOURCE);
 END TRACE FUNCTIONS
END APPLICATIONS
```

In the example, vardisplay, trace, and changedvar are auxiliary functions followed by their parameters. Parameters may be names of selected semantic variables (env and state), build-in standards which provide communication with the environment (INPUT, SOURCE, RESULT) or the results of other basic functions. The latter possibility is necessary in order to provide communication within complex debugging functions which are implemented by a combination of several basic functions. For each of the three debugging functions specified above, a corresponding menu item will be generated which may be selected by the user and will trigger its execution. The display function is directly executed by calling vardisplay with the actual environment and state and the user's input. vardisplay first analyses the input to obtain the variables to display, then retrieves the current value of the variables form the environment/state mappings and writes the resulting information in condensed form in the second window. Selecting one of the trace functions has no immediate effect. The corresponding functions are called whenever the execution of a statement or an assignment is completed. The changedvar function is used to print the old and new value of the variable changed by the assignment (here, state is the state *before* execution of the assignment, whereas RESULT contains the result of the statement execution, i.e. the state *after* execution of the assignment.

A language definer may use the simple stand-alone display and trace functions to implement a quite powerful debugging system following predefined strategies. If more complex functions are desired he may combine basic functions but then there will be no predefined coordination of these functions.

In order to initiate more complex functions or to ask a user for special debug parameters (conditions, number of steps etc.), so-called *continue functions* may be specified. *Interrupt functions* can be used to terminate such a function. Furthermore, several actions at different nodes may be specified: *Entry functions* will be executed when control reaches a node and *result functions* when control leaves it. The following example introduces a complex test-function, which specifies a n-step-function for statements. (The complete definition can be found in the appendix).

```
DISPLAY FUNCTIONS
   initstep:  (askstep initstep);
END DISPLAY FUNCTIONS
INTERRUPT FUNCTIONS
   itrstep:   statement -> (breakstep nstep);
END INTERRUPT FUNCTIONS
ENTRY FUNCTIONS
   nstep:     prog,
              statement -> (countstep initstep,
                                 itrstep, nstep);
END ENTRY FUNCTIONS
```

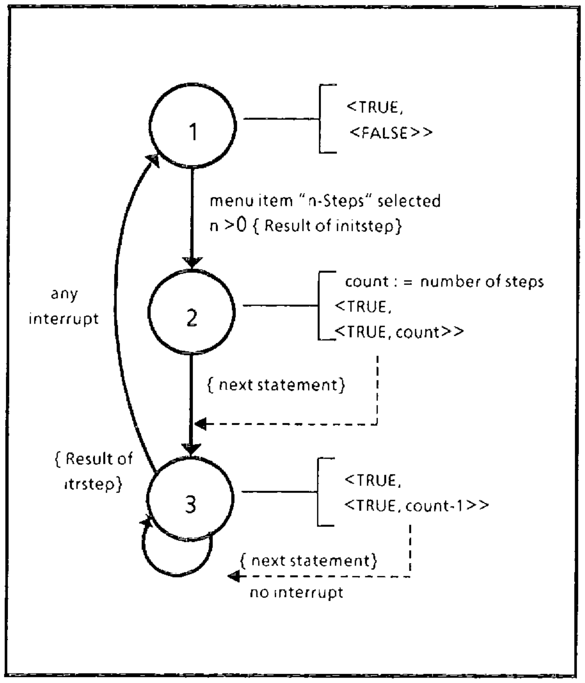
When control flow reaches the meaning function of any statement three actions will be performed by the debugger in general: First, interrupt functions are evaluated, second, when an interrupt occurs or a breakpoint has been set the user is asked to select one or more of the displayed menu items, and third, entry functions are evaluated.

In order to implement the n-step, three basic functions have to be combined. One function will initiate the n-step, one will terminate it and at least one is necessary in order to organize the work. For coordination purposes the results of the basic functions are available for other basic functions: if a function name is specified in the parameterlist of an application, a pair of a system information and the result of the last evaluation of this function will be assigned to that name. The system information is a boolean value which becomes true for:

- interrupt functions: when an interrupt occurs
- display or continue functions: when the associated menu item has been selected
- all other functions: when they are evaluated at the first time so that they are in a defined state.

Interrupts and selections will be reset automatically by the debugger after the evaluation of the entry functions. The n-step-function is build of initstep, a display function to start the testfunction; nstep, an entry function to initialize the test and to count the steps; and itrstep, an interrupt function to terminate the test. The work is done by countstep which gets the initial value of the stepcounter from the result of initstep and decrements it at each statement. The result of countstep is a tuple, whose first element is a boolean value which signals, that the n-step-function is active and the second component is the current value of the stepcounter.

nstep becomes active, when the n-step item was selected, and inactive, when an interrupt occurs, especially, when itrstep has forced it. The following diagram shows the state-transition of the function nstep.



4. Implementation issues

Executing a program within the PSG system is performed in the following way. First, the abstract syntax tree of the fragment to be executed is transformed to a corresponding term of a functional language. This functional language is the basis of the denotational semantics definition and is an extension of type-free lambda calculus. The semantics definition is used to generate a language-specific compiler performing the transformation mentioned above. Second, the compiled fragment is executed by the standard PSG interpreter (note that the interpreter itself is language-independent). The interpreter uses a call-by-need reduction strategy for evaluation of the compiled fragment (see [2] for details).

During compilation of a fragment correspondences between the abstract syntax tree and the terms of the functional language are established in both directions. To all terms belonging to the meaning function of a certain tree node, a pointer to that tree node will be assigned. From the tree node, another pointer refers to one distinguished term of the meaning function. The editor kernel maintains links between the abstract syntax tree and screen positions. The connection between screen positions, abstract syntax tree and terms is used to implement (among others) the breakpoint and trace mechanisms of the debugger in a language-independent manner. No tree pointer is assigned to terms belonging to auxiliary functions.

The interpreter and the debugger are distinct program parts running as coroutines. After compilation of the fragment, execution is started by calling the interpreter. Prior to the evaluation of any term, the interpreter looks at the tree pointer attached to the term. If this tree pointer is defined and if it is different from the last tree pointer encountered by the interpreter in previous evaluation steps, the debugger is called by the interpreter. Thus, the debugger is called whenever evaluation changes from a term belonging to the meaning function of one tree node to a term belonging to the meaning function of another tree node. (The only other possibility to call the debugger is when an asynchroneous interrupt, such as pressing a special function key, occurs).

However, the main activities of the debugger are only activated when evaluation reaches one of the above mentioned distinguished terms of a meaning function. These terms, called *entry terms*, serve as a representation of the complete meaning function derived from a tree node. This distinction is necessary, since most debugging activities should only once be activated for one tree node. Entry terms are identified by comparing their tree pointer with its attached term pointer. Entering the debugger when evaluation reaches a non-entry term, gives the debugger the possibility to change the current selection (by redrawing the inverted line on the screen). If evaluation encounters an entry term, the debugger is called twice: once before the evaluation of the entry term (i.e. the meaning function), and once after the evaluation of the entry term is completed. The debugger processes entry terms in the following way:

- Before evaluation of a meaning function
  - push nesting stacks
  - push selected semantic variables
  - evaluate interrupt functions
  - enter interrupt state when a breakpoint is reached or an interrupt-function yields 'true'
  - evaluate entry functions

- After evaluation of a meaning function
  - evaluate result functions
  - evaluate trace functions
  - pop nesting stacks
  - pop selected semantic variables
  - update selected semantic variables

Basic functions (which are terms of the functional language) are evaluated by calling the interpreter from the debugger. Since terms which belong to basic functions have no tree pointers attached, calling the debugger recursively is avoided. A data structure called the *global environment* is used by the debugger, it contains copies of the current values of all selected semantic variables, the current values of the built-in variables like INPUT, RESULT and SOURCE, as well as the result values of the last call of all debugging functions.

5. Experiences with the system

Several approaches have been published which describe how to generate compilers from denotational semantics (e.g. SIS [5], PSP [8]) or how to integrate execution tools based on denotational semantics into generators for interactive environments (e.g. GANDALF [1], PoeGen [7]). Until now, however, nobody seems to have implemented a generator for interactive debugging facilities within such a system. We feel that doing so is possible within the scope of systems like GANDALF and the Synthesizer Generator (in fact, the Synthesizer Generator has been used to produce an interpreter and debugger based on attribute grammars for a Pascal subset [6]). However, knowing the specification languages of these systems, it is our impression that an implementation effort is quite cumbersome. Contrarily, we think that the PSG debugger specification language is more compact and more easy to use.

Until now, we have generated debuggers for Pascal and Modula-2. It turned out that the specifications are not very long: the specification of the Pascal debugger needs less than 300 lines of specification language (where most of the specification deals with processing the user's input and the production of readable output). We also believe that the generated debuggers are powerful and easy-to-use tools; this is especially true if a personal workstation with bitmap display and mouse is used.

There are, however, some limitations within the current implementation. First of all, the run-time performance of the generated interpreters and debuggers is not overwhelming. This problem is however not mainly due to the debugging system, but due to the interpreter kernel and especially the complexity of denotational semantics definitions. Compiling the terms into (perhaps abstract) machine code makes execution much more faster. In this case, however, the interaction between the machine code (which represents the interpreter) and the debugger has to be redesigned.

Our experiences with the generation of debuggers for Modula-2, Pascal (using different semantics definitions) as well as Lisp has shown that the usability of the specification mechanism depends largely on the structure of the semantics definition. For example, our Lisp semantics defines a translation of Lisp programs into some intermediate code, which is then evaluated by an auxiliary function. Because the execution of a Lisp program is primarily done within the auxiliary functions, debugging functions are only active for a short time during the translation part of the execution. This problem is due to the automatic selection of entry terms (done by the generator); similar problems occur with the Modula-2 continuation semantics. To overcome these problems we are currently extending the specification mechanisms, so that the language definer may explicitly select the entry term within the meaning functions.

The current system allows neither modifications of programs during execution nor features like reverse execution. This is due to the fact that interpretation within PSG is actually a reduction process: functional terms are reduced to produce smaller (result) terms; the original terms can never be reconstructed and there is no possibility for backtracking.

6. Final remarks

We think that is possible to generate powerful debuggers from compact specifications. The system is in operation on SIEMENS machines since June 1986; portation to UNIX-based personal workstations with bit-map and mouse is in progress. Although there is still work to be done, we think that we have demonstrated the feasibility of our approach and that the idea of generating all language-specific aspects of an environment has again turned out to be a fruitful paradigm.

This work is supported by the Deutsche Forschungsgemeinschaft, grant He1170/3-1, and by the Bundesministerium für Forschung und Technologie, grant ITR8502C/8.

7. References

[1] Ambriola, V. and C. Montangero: Automatic Generation of Execution Tools in a GANDALF Environment. The Journal of Systems and Software 5,2 (May 1985), pp. 155-171

[2] Bahlke, R. and G. Snelting: The PSG System: From Formal Language Definitions to Interactive Programming Environments. ACM TOPLAS 8,4 (October 1986), pp. 547-576

[3] Bahlke, R. and G. Snelting: Context Sensitive Editing with PSG Environments. Proc. of the International Workshop on Advanced Programming Environments, Trondheim, Norway, June 1986. Springer, LNCS , to appear

[4] Moritz, B. : Entwicklung und Einrichtung eines Testsystems innerhalb eines Programmiersystemgenerators (PSG): Zur Definition sprachspezifischer Testum-gebungen. Diploma Thesis, Technische Hochschule Darmstadt, Fachbereich Informatik, June 1986.

[5] Mosses, P.: SIS - Semantics Implementation System. Reference Manual and User's Guide. Report DAIMI MD-30, Aarhus University (August 1979).

[6] Mughal, K.A.: Control Flow Aspects of Generating Runtime Facilities for Language-Based Programming Environments. Proc. IEEE Computer Society Conference on Software Tools. New York, 1985.

[7] Pal, A.A. and C. Fischer: EDS: Executable Denotational Specifications for Integrated Programming Environ-ments. University of Wisconsin-Madison, 1986.

[8] Paulson, L.: A Semantics-directed Compiler Generator. Proc. 9th ACM POPL, Albuquerque (Januar 1982), pp. 224-239.

[9] Seidner, R. and N. Tindall: Interactive Debug Requirements. Proc. of the ACM Software Engineering Symposium on High-Level Debugging. ACM SIGPLAN Notices 18,8 (August 1983), pp. 9-22.

Appendix

Definition of the n-steps function:

```
DISPLAY FUNCTIONS
    initstep: (askstep initstep);
END DISPLAY FUNCTIONS

INTERRUPT FUNCTIONS
    itrstep: stat -> (breakstep nstep);
END INTERRUPT FUNCTIONS

ENTRY FUNCTIONS
    nstep:  prog, stat ->   (countstep initstep, itrstep, nstep);
END ENTRY FUNCTIONS

GLOBALS
    askstep =  LAM dummy.  ANSWER 'number of steps ? '; ASK INT;
    countstep   =  LAM srq, itr, act.
                        { srq = last result of initstep }
                        { itr = last result of itrstep }
                        { act = last result of nstep }
                        IF SELECT srq, 1        { corresponding menu-item selected ? }
                        THEN
                            LET init = SELECT srq, 2 IN  { initial stepcount }
                                IF INTEQU init, 0
                                THEN
                                    < FALSE >          { stepcount = 0  = > inactive }
                                ELSE
                                    <TRUE, init >      { activate nstep }
                        ELSE
                            IF SELECT itr, 2
                            THEN
                                < FALSE >          { interrupt occured = > deactivate nstep }
                            ELSE
                                IF SELECT act, 1  { function ever evaluated ? }
                                THEN
                                    LET state = SELECT act, 2 IN
                                        IF SELECT state, 1      { nstep active ?  }
                                        THEN
                                            < TRUE, INTSUB SELECT state, 2, 1 >
                                                    { decrement stepcount }
                                        ELSE
                                            < FALSE >
                                ELSE
                                    < FALSE >;
    breakstep =     LAM act.
                        { returns true if interrupt should occur  }
                        IF SELECT act, 1   { nstep evar evaluated ? }
                        THEN
                            LET state = SELECT act, 2  IN
                                IF SELECT state, 1                  { nstep active ?  }
                                THEN
                                    INTEQU SELECT state, 2, 1 { final stepcount ? }
                                ELSE
                                    FALSE
                        ELSE
                            FALSE;
END GLOBALS

TITLES
    initstep -> ' n-Steps'
END TITLES
```